Good Morning, Everyone

Let us get started. Thanks for you to attend this topic: best practice on ECS flink platform: SMaRT SRE Platform (2.0).

This is the first time we use ECS and Flink Service to redesign our SMaRT system, I also think it is the first time to use Flink Service to deliver

Response as soon as possible in citi.

Since last August, we begin to design SMaRT system to meet the ICG level requirement, and we official publish the full new SMaRT System in this June. from the POC to real implementation, it takes us almost 9 month, we are experiment how to enhance high availibity , high performance and keep the data consistence for Smart system.

This is today's agenda

1. Simple introduce the SMaRT system.

2. Introduce the whole architecture, in this part, I also convert HA, H performance, data consistence.

3. The deployment profile, include the performance results.

4. Summary

SMaRT Introduction

SMaRT is short for Surveillance, Monitoring, and Regulatory Tool, SmaRT provide centralized interface and implementation for pre-trade surveillance, regulatory and compliance checks. You can also consider SMaRT as a rule engine which can have an ability to run the executable rules. Our client like pre trade system, order management system, trade capture system can use Smart to verify the trade is legal or not. In smart, we already support rules: PBC, Brexit, QFC, and so on

Smart provide easiest ways to communicate with client application system. Via restful api. our client can make a SMaRT request and use http protocol to send the request to SMaRT, SMaRT will use its owned service to produce response, which include suggested action to indicate the trade can be booked or not.

Smart support two kind of message format: XML and json, client can use any of the form to submit the request. here this is the simple example for SMaRT request and response, the left one include client application information and required rules, and right one is related to trade information. The below one is SMaRT response, we can see the result is hard-override so that trade cannot booked, trader also can get useful hint from the message field.

This is the simple introduction Smart

Next part will cover the whole architecture to implement SmaRT system. As I introduced in the beginning, we already have an old SMaRT system, we can called it SMaRT 1.0. It is still providing service. in citi, migration is not easy to do. SMaRT is not a system, it contained Smart rules, and these rules are executed in Factory service. These rules need to be coded and saved into Oracle. Factory contains three parts. dl engine, which is core part of factory, factory provide a capability to run dynamic report, and report is write in dl language(dependency language), and also SmaRT rules are written in the DL. You can think it is a compiler, which include compile dl source code and execute the source code to produce results. SMaRT have no direct interaction with business system, so DL iterator is designed to get the data from third party system, like core team's redqueen and oracle database. We also use global cache to accelerate the data fetching. Smart 1.0 is single service, we can think it as Monolithic architecture, and all the submodules are packaged into one service.

in new architecture, we use micro service concept to design SMaRT SRE system(2.0), the whole system is running on ECS platform, all the service is deployed into ECS platform by light speed. The whole system is spited mainly into four parts, orchestration, we can also called access service (Rest service), which provide restful api to accept the client request, computing service, it is packaged into flink service. Why we choose flink service as our computing service, the flink is a streaming platform, which can support a multiple operations in parallel. And our SMaRT requests can be split as many smart rules, this rule can be executed separately on each flink node, after all the small rule is executed and they are aggregated one response by flink. It is suited for parallel computing concept. And fink also provide configure parallism. If we think some flink operations are time consuming, we add assignment more recourse to the operation to reach our needs. At the current time, the most efficient communication channel between orchestrator and flink service, is Kafka service, we use gsp core team's Kafka to bridge the front end service and flink service. the next sub services is our cache service, which is also deployed into ECS, redis is really high performance component and can obvious reduce the whole system's response time, the last component is mongo service, which provide the auditing and report functionality, we save all the request and response into mongod db for latter auditing, and we also have ui interface to provide metric summary, all the data is fetching from mongo service.

we can simply abstract the SMaRT 2.0 as the following components; rest service, which provide access capability, rule service(flink service) provide rule check capacity, mongo service provide tracing and auditing capacity, and redis service ,which providing millisecond cache.

to summary, we draw the comparison between SMaRT 1.0 and SMaRT 2.0. Smart 1.0 use Physical server to deploy service, it is monolith service , all the component is formed into one service, and use factory to run the rule check, it use ML and Oracle to save data. it also use infinispan as cache , while in SmaRT 2.o, it is hybrid architecture, it is major in ECS,plus 8 VM. it is micro service architecture, and it is use flink to run rule check. The data is saved into mongodb, and Oracle, it use data center levels cache. for high availability, it is very high for Smart 1.0 it deployed into several nodes and use load balance to route the traffic to backend, so it has no single point filature issues, but for Smart 2.0, it is powerful and we enhance the HA in many factors not only load balance, we will cover it in later section. For performance, Smart 1.0 is serial rule engine, all the rules in the request is executed one by one. And in Smart 2.0 rules can be split and executed in parallel.

The next part is covered HA enhancement.

We use duplicate theory to enhance the whole system's HA, in one data center, we set up dual flink service (flink one and flink two). They are not belong to master and slave architecture. They are working to provide the rule check at the same time. rest service have multiple pods so the load balance route the coming request to any of the pods, the request will be delivered into Kafka service, and both of the flink service will get the request and process it in their own cluster, and they both deliver the response to Kafka service. And rest service will return the first one to client. Which means that faster response will send to the client. We have no single flink service failure issues.

To consider more, we still think maybe it is still not safe that all the service is deployed into ECS, we also design COB solution, we have another plan that is to provide service into VM, we have factory service which can run SMaRT rule, and just make simple change and bridge the rest service with factory service. So our service still can process client's request in the failure of cluster issues.

we enhance the HA by setting up two ECS cluster in one region, and two ECS cluster and vm cluster is composed to provide the service by WIP. We have two WIPs, in name, we have one, in EMEA, we another one. The down time in single cluster is not an issue in SMaRT. For example, we set up two clusters in name, when 390 is out of service by patching, or network. SMaRT still can provide the service in ruth. We have not seen two ECS clusters are down at the same time after releasing about two months. By the one way, even if two ecs clusters are both, we still can hand it and we have VM solution to provide service.

The next part for ha, we depend the Kafka service, and four ECS cluster connect the different Kafka service. In EMEA, message bus team help to set up two new flinks to support SMaRT application. We also consider the case when one Kafka service is out of service, our SMaRT can use another one by manual failover. In the SMaRT release time, Kafka service is not supported the failover mechanism. So the failover is provided by application failover. When we detect the outrage of Kafka service, we will ask FO support to run an autosys job to do the failover switch.

For the performance side. We fully use the flink service' capability. We know if we want to enhance the system performance, we often need to split big tasks, or long running task into several sub small task. New SMaRT platform. This is smart operation graph, the smart request will splits by entity, and then by rule. And these rules will be executed by parallel, that that we will aggregate the response by rule and then by entity, and finally generate a single response, and put it into response Kafka topic.

We also leverage how the parallism affect the system. paramllism like ecs replicate, if set the parallism to 5, we will have 5 node to process the request in parallel. flink service almost take 90 & resource in one data center. we know that rule check operation is time consuming operation, we assign more resource to that opeation.

Another side, I want to share, random parallism for link operation will harm the performance. Communication across pod also will delay the performance. if the set the same parallism for neighbor operation, flink service will group this operation into one node. They will have no network overhead. We need to decrease the pod communication. Because flink service set delay mechanism in order to reach maximum throughout. The request is processed by current operation, the produced result will not send response to next node until buffer is full or the time out is coming.

The second performance tuning is about taking fully use of Kafka partition. Kafka partition number is also related to parallism. We have producer to send the request and use consumer is fetch the request. And single Kafka partition is not shared by backend consumer, and consumer can take two or more Kafka message. If the parallism is too small, and messages will be accumulated in message bus. If one message is timeout, the latter request will be timeout. So we need to adjust paramllism number according to requirement. Another side, we want to cover is response. How does the rest service get the response? If every pod get all the response and filter its response, it is time consuming. We let our rest service just fetch special Kafka partition, for example, rest service send the request through the partition 1 and he also get the message from partition 1. It makes rest service very light weight.

The third party for performance: taking fully use of network distance. The network distance is a little long in citi, it will not acceptable if the communication happened across data center especially for real time process system. This year, message team set up two new clusters in EMEA, and requeeen also set a two data centers in EMEA. We also setup data center level's regional cache service. Each data center, they only access its service. So we have 4 ecs data centers, name will contact name's service, and EMEA will contact name's service.

By doing that, SMaRT application can provide real time delivery capacity. About 50 percent will be processed in 50ms, and 99% will processed in 300ms, I will share the performance result.

Next part is the data consistence.

SMaRT does not produce any business data, but it need to access the business data for rule check. Lucky, most of the data is persistent in RedQueen, this is the data flow from business to redqueen, and every node will access red queen in flink service, in order to improved system performance, we use redis cache to cache accessed business data. So the node will first fetch data from the redis cache, if the data is hit in cache, just use the data, and if not exist in cache, node will access the red queen. That is the for the data cache.

We have an issues, in this design, we are not sensor that that data is changed by the business. in the cache we may have the dirty i . We design fully new cache flushing mechanism. In compatible with factory mechanism, we add a new cache flush overflow in the flink service. We designed a lot of cache reports and use autosys to schedule to flush the data. According to the bussines, we can run them every 5 minutes, or every 1 hours. We have common client library which can trigger to send the report update request.

We have another requirement, we need to real time update. From AMC account ,we need to check the account is legal in real time, luckily, data acquisition team have already have one, they will publish the update to the Kafka topic, we designed a new service to accept the update and update the cache in real time.

Another important things: how to maintain the report in flink service. It is not acceptable that we begin to parse the request and run the rule when the request is coming. Because the time to parsing the source report is time consuming. We need to parse the rule into local cache in every flink node in advance. We also support the real time update for the rule to meet the urgent change or bugs in rule. We can use smart dashboard to change the rule and the write the updated timestamp into cache. And every node is schedule to get the updated timestamp and to check whether a new rule is added or some rule is changed every 60 seconds, if yes, we can parse the rule in advance.

That is all the data consistence in Smart.

At now, I have illustrated all the architecture of Smart.

Next part is about deployment plan, or how we use the applied resource to deploy SMaRT service.

The table is the resource capability. In name, we use the same resource for 390G and RUTH, we have 175 cpu cores and 1400 G byte memory, while in the EMEA, we have almost half resource, we have 90 cpu cores and 1000 G memory.

Except that we have 8 VM, each node have 8 cores and 64 G memory.

For the resource allocation for each data center, here I use 390G as an example, we allocate 12 pods for rest service, each pod have 8G memory and 1 cores. We have two flink services. Every services we have one job manager and 65 Task Manager. They have same configuration like Rest service. Another important component, which is Redis cache. We use cluster mode, cluster compose 6 pods, and 3 masters and 3 slaves mode. We give them biggest cpu allocation (1.5).

We also have auxiliary service, like mongo service and console service, redis manager, which can help us to track, monitor system performance. After that configuration, we also use 90% resource.

Here is the performance test.

That is all for today’s sharing:

Any other question before end up

Thanks for your time again.